[NASA-CR-177952

NASA Contractor Report 177952

NASA-CR-177952
19850026063

MODULAR DIGITAL HOLOGRAPHIC FRINGE DATA PROCESSING SYSTEM

James G. Downward, Pamela C. Vavra,
Frederick S. Schebor, and Charles M. Vest

KMS FUSION, INC.
Ann Arbor, Michigan

Contract NAS1-17945
June 1985

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NF00710

1 Report No	2 Government Accession No	3 Recipient's Catalog No
NASA CR-177952		

4 Title and Subtitle	5 Report Date
Modular Digital Holographic Fringe Data Processing System	June 1985
	6. Performing Organization Code
7 Author(s) James G. Downward, Pamela C. Vavra, Frederick S. Schebor and Charles M. Vest	8 Performing Organization Report No
	10 Work Unit No
9 Performing Organization Name and Address KMS Fusion, Inc. 3621 South State Road, P. O. Box 1567 Ann Arbor, MI 48106	11 Contract or Grant No NAS1-17945
	13 Type of Report and Period Covered Contractor Report
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546	14 Sponsoring Agency Code 324-01-00-01

15 Supplementary Notes Final Report
James G. Downward, Pamela C. Vavra, and Frederick S. Schebor: KMS Fusion, Inc.
Charles M. Vest: University of Michigan, Ann Arbor, Michigan
Langley Technical Monitor: Jag J. Singh

16 Abstract

This is the final report summarizing the work done under Small Business Innovative Research Program Phase I Contract No. NAS1-17945. Under this contract KMS developed and tested a software architecture suitable for reducing holographic fringe data into useful engineering data. The results, along with a detailed description of the proposed architecture for a Modular Digital Fringe Analysis System,are presented in this report.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
Interference Fringe, Automatic Fringe Data Processing System, Expert Decision Module	Unclassified - Unlimited Subject Category 35

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price
Unclassified	Unclassified	53	A04

For sale by the National Technical Information Service, Springfield, Virginia 22161

Table of Contents

Table of Contents..... 1
Project Summary..... 3
Project Objectives and Overview of Results..... 5
Details of Phase I Research..... 6
Technical Background..... 6
Fringe Analysis System Architecture..... 7
1 Introduction..... 7
2 The FAS Architectural Description..... 8
3 The FAS Knowledge Architecture.....10
4 Fringe Processing Module Architecture.....12
4.1 Fringe Processing Command Language.....13
4.2 Fringe Processing Module Device Independence.....14
4.3 Fringe Processing Module Knowledge Interface.....14
4.4 FAS Menu Processing Module Support.....14
5 Expert Decision Module Architecture.....15
5.1 Expert System Development Language.....15
5.2 Evaluation of a Rule-Based Approach to Fringe Analysis.16
5.3 Expert Decision Module Design Requirements.....18
Establishing Requirements for a Fringe Analysis System.....19
Evaluation of Fringe Analysis Algorithms.....20
Validation Studies of the Proposed FAS Architecture.....23
References.....25

Appendix A:
Functional Description of Analysis Shell Operation.....A-1
Appendix B:
Detailed Description of the FAS Monitor Program.....B-1
Appendix C:
FAS Inter-Object Communication Architecture.....C-1
Appendix D:
Named Knowledge Architecture and VMS Logical Names.....D-1
Appendix E:
Fringe Processing Command Language Architecture.....E-1
Appendix F:
Implementation Considerations for Expert Decision Modules..F-1

Report Documentation Page

N88-34376

This Page Intentionally Left Blank

Phase I Project Summary

Holographic and interferometric techniques are now used routinely for measuring wind tunnel flow field density distributions and structural deflections. For these measurement techniques to realize their full potential, digital systems which can automatically process the photographic fringe data into useful engineering data need to be developed. The objective of this research was to develop an architectural design for a general purpose fringe analysis system which would be able to utilize expert knowledge to assist in automatically analyzing fringe images.

Fringe analysis is generally a very time-consuming process requiring continual human judgement. To improve accuracy, eliminate drudgery for highly skilled engineers or technicians and, most important of all, to allow the analysis of vast amounts of data needed to understand complex or time dependent phenomena, there is an obvious need for computerized fringe analysis. For this reason, NASA sponsored a Workshop on Automated Reduction of Data from Images and Interferograms in January 1985. To date, what work has been done in this field has usually involved writing large fringe-tracing codes of varying degrees of sophistication which are generally designed to operate in the context of a particular experiment. Thus, there is a need for two things:

- A higher degree of automation of fringe analysis than currently exists, and
- A software system which can absorb and use knowledge about and techniques for fringe finding as they are developed or modified for new tasks.

The current research addresses both of these matters simultaneously and aims to produce a software system capable of operating in a stand alone manner or of encompassing all existing and any future fringe finding algorithms in a user-oriented manner. It thus becomes a useful tool for anyone involved in fringe analysis regardless of whether they need a new standalone system, or a system within which to implement a new approach to some aspect of the analysis.

During the study, the problems inherent in automatic fringe data analysis were studied. Based on this study and in-house experience in analyzing fringe data, an architecture for a fringe analysis software system was developed. The proposed system would:

- Provide the framework required for an automatic fringe data processing system.
- Process the fringe data in discrete steps using mono-function processing modules.
- Share knowledge gained at any processing stage with subsequent processing stages.
- Utilize expert knowledge to select fringe processing algorithms and control the processing steps.

The proposed design for a fringe analysis system was evaluated by implementing and testing a subset of the architecture in software and by testing the suitability of a number of fringe location algorithms to wind tunnel holographic fringe data. Further work will implement the full architecture in software, develop fringe processing modules, and implement expert decision modules for controlling the processing steps.

Phase I Project Objectives and Overview of Results

During the Phase I contract period, KMS has developed and tested a software architecture suitable for reducing holographic fringe data into useful engineering data. In this report, the results of this work are presented along with a detailed description of the proposed architecture for a Modular Digital Fringe Analysis System (FAS).

The technical objective of the Phase I contract was:

> To design a software architecture for a Modular Digital Fringe Analysis System capable of using expert knowledge to control the processing and analysis of fringe image data into useful engineering data. The sub-tasks for this contract included defining:
>
> - The requirements for an Analysis Shell.
> - The knowledge architecture for the system.
> - The system data types and structures.
> - The fringe processing modules.
> - The interface to Expert Decision Modules.

The technical goals of this project have been met. Specifically,

- A VAX/VMS software architecture meeting these goals was designed which
 - Uses an Analysis Shell to control the processing of fringe images by single-function processing modules.
 - Allows fringe knowledge obtained using one processing module to be shared with subsequent processing modules.
 - Uses a device independent fringe processing language to interface to the processing modules.
 - Allows fringe processing to be totally controlled by adding suitably designed Expert Decision Modules.
- NASA's needs for fringe analysis were evaluated to insure that the system design would meet NASA's programmatic goals.
- Existing KMS fringe analysis programs were used to investigate requirements for analyzing holographic wind tunnel data.
- Available AI languages and knowledge engineering tools were evaluated for use in developing Expert Decision Modules, and OPS5 was found to have suitable performance for use in fringe analysis applications.
- The architecture was evaluated to insure that it met its design goals of functionality and performance by developing prototypes for the Fringe Analysis Shell, the device independent Fringe Processing Language, and a number of Fringe Processing Modules.

Details of Phase I Research

Technical Background

Holographic interferometry is routinely used to study a wide range of aerodynamic problems. While the primary usage has been for two- and three-dimensional flow field visualization in wind tunnels(1,2) and shock tubes,(3) holographic interferometric measurements also have been used in ballistic ranges, rotor test chambers and turbine facilities.

The primary advantage of holographic interferometry over other measurement techniques is that it combines visualization with a nonintrusive quantitative measurement of the entire density field. In addition, holographic interferometry often can provide a two-dimensional measurement of the pressure and, in some case, the velocity field, and may even be used to analyze dynamic or unsteady flow fields.(4) However, to effectively utilize holographic fringe data, immense amounts of two dimensional fringe image data must be numerically analyzed. Until recently, attempts to address this data analysis problem have met with limited degrees of success.(5,6)

Recently, however, work by Becker et al. (7,8,9) has shown that digital analysis of holographic interferograms can provide aerodynamicists with a new powerful analysis tool and makes possible wind tunnel measurements which would otherwise be impossible. For example, the ability to analyze interferograms in a semi-automatic fashion, makes possible tomographic analysis of the Rotocraft experiment at NASA Ames.(10).

While Becker has demonstrated the usefulness and feasibility of applying digital fringe analysis to a number of aerodynamic applications, considerable work remains to be done if holographic fringe analysis is to find routine use in aerodynamic methodology or is to find commercial applications in other areas such as Holographic Nondestructive Testing (HNDT). The need for additional work was emphasized when in January 1985, NASA Ames and the U. S. Army Aeromechanics Laboratory sponsored a "Workshop on Automated Reduction of Data from Images and Holograms" to address this problem. In a review paper at this conference, Vest summarized the fringe data analysis problem:

> Perhaps the most pressing problem in the field is that addressed in this workshop, namely the automated analysis of interferograms to provide fringe order data. In many applications this presents a formidable image processing problem. Furthermore, in most applications significant interaction with a knowledgeable operator is likely to be required...the problem may be ripe for application of concepts of artificial intelligence, particularly expert systems.

Fringe Analysis System Architecture

1 Introduction

The goal of fringe research at KMS is to develop a packaged fringe analysis system (software and hardware) capable of automatic reduction of a wide variety of fringe data. Developing monolithic software programs to provide this capability was not considered to be a suitable technical approach for several reasons.

First, fringe images can be extraordinarily complex and difficult to interpret. A program correctly working with one type of fringe image may fail with another. Among the conditions which make analysis difficult are:

- o Diffraction by solid boundaries
- o No region of known reference value
- o Very closely spaced fringes
- o Unknown sign of fringe order
- o Nonuniform background irradiance
- o Data blocked by opaque objects
- o Caustics due to refraction and diffraction
- o Extraneous fringes
- o No fringe closure
- o Inadvertent wedge fringes
- o Laser speckle
- o Discontinuous fringes
- o Broad, "cloud-like" fringes

Second, although a conventional analysis program may address some of these problems, the program may have limited applicability for analyzing other fringe image data because the rules or heuristics built into the program may be inappropriate when applied to the new data. Consequently, in order to adequately address the general fringe analysis problem, a more generic approach is needed in which:

- o Knowledge about how to analyze fringe images controls the fringe analysis process.
- o The fringe data are processed in modular, discrete stages which do not make assumptions as to the specific nature or sources of the fringe data.

The Phase I study addressed this problem by designing and testing a **software architecture** for a Modular Digital Fringe Analysis System. The implementation of this approach will allow new fringe analysis problems to be solved in a "building block" fashion. However, before presenting the results of this study, it is important to define what is meant by the phrase "software architecture".

Writing a single Fortran analysis program incorporating some algorithm, or developing a new algorithm for that program is a straightforward and well understood problem. A much harder problem to address is creating a software package of many analysis programs all of which must communicate with each other.

A current software engineering approach to creating a large analysis package is to first create its architectural design. Creating a software architecture is very similar to creating an architectural design for a building. Starting out with an overall concept of the design goal, at

successive stages the software engineer refines the concept with greater and greater detail so as to show how the design's component parts will correctly fit together. Specifically, in designing a software architecture:

- o Goals are established as to what tasks the analysis package should perform and how it should function in relationship to those using it.
- o The required components of the package and their functions are defined.
- o The methods by which elements of the package communicate with each other and the outside world are defined.
- o The correctness of the design may be verified by implementing critical software components prior to committing to a full scale software development project.

In this report, the architectural design for the software of a Modular Digital Fringe Analysis System (FAS) is presented. The primary goal of this architecture is to allow a general purpose fringe processing and analysis system to be developed which is:

- o easy to maintain and support
- o versatile and expandable
- o able to use expert knowledge
- o able to process fringe data automatically.

The architecture for the FAS is composed of four interrelated parts:

- o The analysis shell architecture
- o The knowledge architecture
- o The fringe processing module architecture
- o The expert decision module architecture.

These parts will be discussed in subsequent sections.

2 The FAS Architectural Description

The Fringe Analysis System consists of both the hardware (computer, fringe digitizer, operator terminal, image display, etc.) and the fringe analysis software. The relationship of the software to the to the system as a whole is schematically shown in Figure 1. This study has addressed the architectural design for the fringe analysis software which is the core of the fringe analysis system.

The FAS Analysis Shell consists of a FAS Monitor program under which can run any number of external, independent, fringe processing modules (PMs) and cooperating Expert Decision Modules (EDMs). The Analysis Shell maintains independence from the specific hardware supported by the computer by performing input/output operations via device driver routines in the outermost shell. The input to the system is a fringe image plus any operator supplied input. The output is the processed fringe data in a format suitable for numerical, engineering analysis.

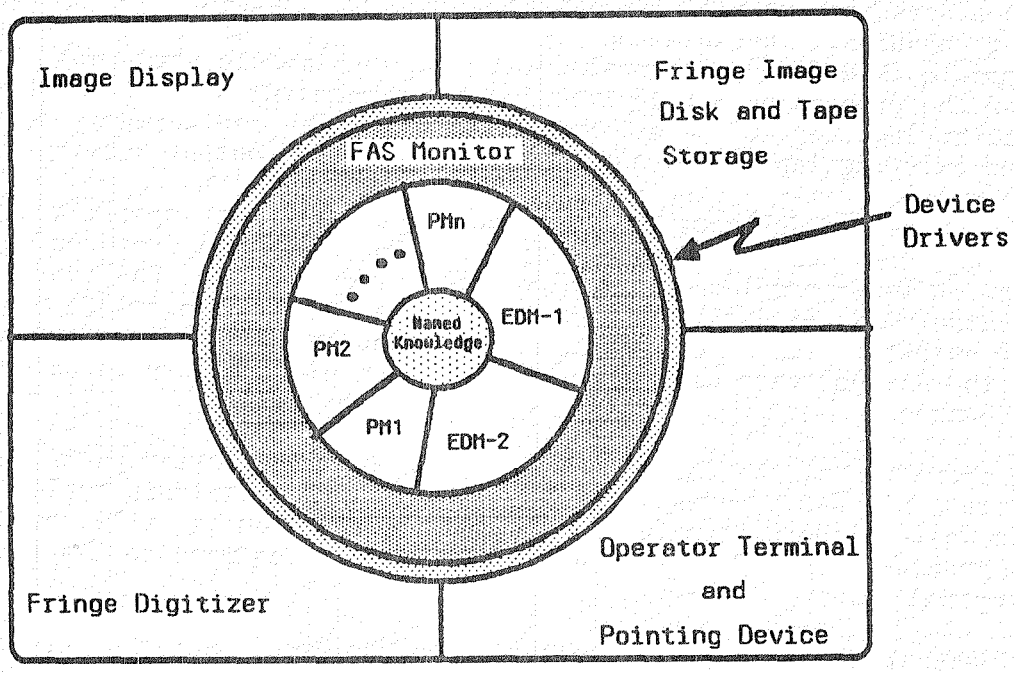


Figure 1. Fringe Analysis System

The Analysis Shell is created and maintained by the FAS Monitor program which

- Manages the communication channels between the operator and the processing modules.

- Maintains global knowledge used by the analysis modules.

- Keeps track of the current state of the fringe data being analyzed so that the cooperating analysis modules stay in synchronization.

Details of the operation of the FAS Monitor and the FAS Analysis Shell are presented in Appendices A-C. Appendix A presents a functional description of the FAS Analysis Shell in operation. Appendix B discusses the multi-level processing architecture maintained by the FAS Monitor. Finally, Appendix C discusses the FAS Communication Architecture.

The FAS global knowledge data base consists of knowledge required for the cooperative analysis of the fringe data by a collection of independent modules. It is composed of information supplied prior to the start of analysis (historical, configuration, etc.), and knowledge generated by the modules during the process of fringe analysis. This knowledge may be accessed by name by any module running within the Analysis Shell. In addition to global knowledge, domain-specific, expert knowledge and heuristics for use in guiding the fringe analysis process is available to Expert Decision Modules. On VAX systems the global knowledge data base is maintained by the VMS operating system in cooperation with the FAS Monitor program.

Each processing module is designed to be independent of all other processing modules. Because the format of the input and output data required by each module is defined, internal details of each module need not be known outside of the processing module itself. Any processing module can be replaced without affecting the FAS operation as long as the replacement module can accept the existing input data and return the expected output data.

The modules are designed to be independent of the specific types of hardware used in conjunction with the FAS. For example a frame buffer, video digitizer, or image display device might be used by the FAS software for storing, acquiring, or displaying the fringe data during the fringe processing. Input and output to these devices is accomplished by low level software drivers so that the FAS software can be implemented on a wide variety of hardware with minimal changes.

The operation of the FAS is straightforward. An operator gives an initial command to the FAS, perhaps to invoke a predefined analysis script (list of things to do). As each item in the script is read from a file by the FAS Monitor program, the FAS Monitor sends off a command to the appropriate processing module.

For some analysis problems, performing a series of actions stored as a script file may be adequate. However, if the fringe analysis problem is complex, the problem might benefit by applying expert fringe analysis knowledge to control the fringe processing steps. In this case, an Expert Decision Module (EDM) may be activated.

The EDM examines the global knowledge about the fringe image being analyzed and decides what actions to take. If it is unable to perform the action by itself, the EDM sends back a command to the FAS Monitor to direct a processing module, to perform that function (Shell Callback). The EDM waits for the processing module to complete its task and update the global knowledge as appropriate and then resumes analyzing the fringe image. When the EDM finally has succeeded in accomplishing its goal, it exits, returning control of the FAS to the operator or script file as appropriate.

3 The FAS Knowledge Architecture

In order for the FAS to control the flow of fringe analysis by independent modules, it must provide an environment for sharing current knowledge about the state of the fringe analysis between the FAS Monitor, the processing modules and the expert decision modules.

Knowledge needed by the FAS may take many forms. It may be global (all modules may use it), local (only one module uses it), historical (results of previous operations), situational (describes the current state), scratchpad (use and discard), or expert (for the EDMs).

The emphasis on modularity in the FAS Architecture imposes a number of conditions on the methods used to provide global knowledge to the processing modules.

- Because the FAS is to be modular and expandable, global knowledge must be accessible by name from any module within the Analysis shell. Such knowledge is referred to as **"Named Knowledge"**.

- Because the FAS Monitor, the processing modules, and the Expert Decision Modules are discrete programs, the global knowledge base must be stored external to these programs.

- Because each module activated may need to access Named Knowledge data frequently, access to the knowledge must be as rapid as possible so as not to degrade the over-all performance of the FAS.

- Because many different types of knowledge need to be stored (text, real numbers, integers, etc.) the knowledge storage format must be flexible.

An investigation of a number of possible approaches showed that the proper approach depended on the storage requirements for the given data element. If the information entity to store is physically small in size (< 256 bytes) and need not exist after the analysis has been performed, storing the data as text strings in VMS Job Logical Names was chosen as the best general solution to the problem of providing rapid, random access, by name, to data of widely varying formats. However, if the data is a fringe picture, vector array, or a large quantity of data, or if the data must exist for the next FAS analysis problem, then the data itself is stored as a disk file whose file name is pointed to by a Named Knowledge element. In Appendix D, VAX/VMS Logical Names and their use in storing knowledge of varying formats is be discussed in detail.

Each Named Knowledge element consists of two parts as shown in Figure 2. The first part, the Knowledge Name, consists of a unique name (identifier) of 1-255 alpha-numeric characters. Using the Knowledge Name, one can retrieve the Knowledge Value which will be returned as a string of 0 to 255 ASCII characters. The Named Knowledge data base consists of all the variable length Named Knowledge elements which currently have a non-null Knowledge Name.

The functionality of this implementation technique was evaluated in two ways.

- o The prototype monitor program was used to pass information between the Shell, the processing modules, and the operating system via Named Knowledge elements. The technique proved to be both powerful and simple to use.

- o The speed of retrieving Named Knowledge stored in the Logical Name tables was measured. Independent of table size, the average time for random retrieval of each knowledge element is about 2.5 milliseconds on a VAX-11/750 which is far faster than a disk-based retrieval system.

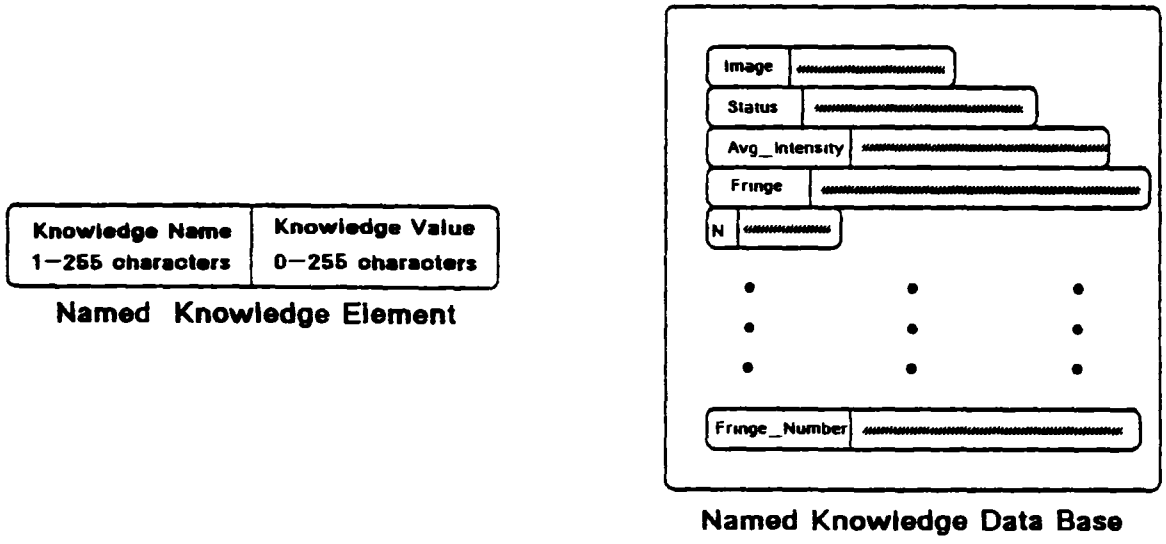


Figure 2. Named Knowledge Representation and Data Base

4 Fringe Processing Module Architecture

The FAS supports fringe processing modules in both the image analysis and engineering analysis domains. By breaking up the fringe analysis problem into a number of discrete steps with well defined goals and end states, fringe processing can be accomplished by a sequential series of commands to process the fringe data.

To support analysis in a modular series of steps, a Fringe Processing Module Architecture has been designed to meet the following requirements.

- Each processing module is a separate program designed to perform a generic class of operations on the fringe data.
- Processing modules are command driven. To perform a fringe analysis function, the appropriate command is constructed and sent by the FAS Monitor to the appropriate processing module.
- Fringe analysis modules may be invoked independently of the Analysis Shell to facilitate program development and testing.
- A logically consistent, English-like, Fringe Processing Command Language is used to send commands to the processing modules.
- The Fringe Processing Command Language and the processing modules themselves possess a high degree of device independence to minimize the impact of special hardware requirements and to simplify the operator's use of the FAS.

4.1 Fringe Processing Command Language

To control the fringe processing modules in a consistent and intuitive fashion, a Fringe Processing Command Language (FPCL) has been designed and tested to support the Fringe Processing Module Architecture. In Appendix E, the preliminary design for the FPCL is presented. In Figure 3 is a diagram of the command and I/O flow supported by the FPCL.

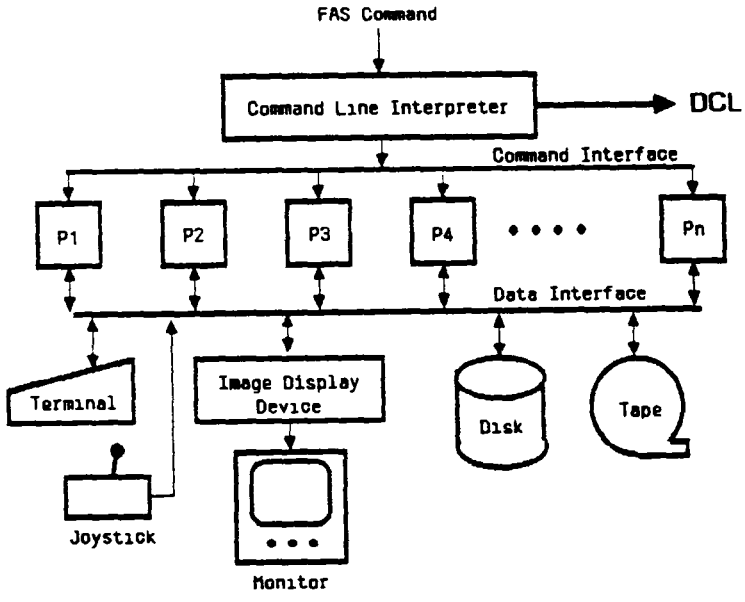


Figure 3. FPCL Command Flow

Because the FPCL follows the general Command Line Interpreter syntax format which is supported by the VAX/VMS operating system, the FAS Monitor is able to make extensive use of system services to simplify parsing and interpretation of FAS commands. First, the FAS Monitor parses the command to see if it is an external VMS command, an internal FAS Monitor command, or an external processing module command. If it is an external VMS command, it is sent off to the VMS DCL monitor for processing. If it is an internal command, the FAS Monitor executes it. If it is an external processing module command, the FAS Monitor uses standard VAX/VMS operating system services to parse the command to insure that it is syntactically correct and that all required inputs are present and to supply any default values which may be needed by an processing module.

If the FPCL command is correct, the proper analysis module is activated, retrieves the parsed command, takes the action specified, and performs any required input/output operations. If the FPCL command is incorrect, an error message is displayed explaining the problem.

The general format of a FPCL command is:

COMMAND[/switch2][...][/switchN] Command_Line

The various command switches may take values and the Command_Line may contain any information as required by COMMAND. Command Switches qualify or select which feature of a given command will be applied. A FAS command may also be issued directly by the user from DCL (i.e., outside of the Analysis Shell), the command must be directly preceeded by a "FAS/". Examples of the FPCL are seen in Appendix E.

4.2 Fringe Processing Module Device Independence

The FPCL and the processing modules are designed to support device independent analysis. FPCL commands may either operate on an image in the FAS local memory or on an image stored on a disk file.

Images may be on a disk file, a frame buffer, or in a VAX memory buffer. The display device may be a frame buffer or an image processor. By performing display I/O via replaceable device interface subroutines, reconfiguring the FAS to use a new device only requires that the appropriate device interface routines be implemented.

4.3 Fringe Processing Module Knowledge Interface

Knowledge is passed two ways in the FAS. The first form of knowledge used by a processing module is information which tells that module what do do next. This information is passed to the module by the FAS Monitor as a parsed command and interpreted by the module using standard VAX/VMS subroutines.

The second form of knowledge is information which tells the module what we know about the current problem. This knowledge is stored as "Named Knowledge" elements and may be retrieved or updated by a module using the Get_Named_Knowledge and Put_Named_Knowledge subroutines.

4.4 FAS Menu Processing Module Support

The flexibility of the FAS Architecture allows for either direct operator control of the FAS (via direct FPCL commands) or for creating tailored operator interface programs to perform selected functions.

At the direct command level, the FAS will perform any syntactically correct command either entered by the operator or from a command script. However, for situations in which operator control is needed but requiring the operator to know the FPCL commands is not desireable, it is straightforward to create a FAS Menu Processing module to sit as an interface between the operator and the system.

The FAS Menu Module could present the operator with one or more menus from which to make selections. Based on the selection input by the operator, an FPCL command to the processing modules would be formed, and passed to the Monitor. A FAS Menu Module, controlled via operator input, performs the same function as an EDM except that for the EDM the expertese resides within the EDM rather than within the operator.

5 Expert Decision Module Architecture

For all but the simplest fringe images, current analysis methods require knowledge (guidelines, processing step order, rules, heuristics) about the methods to be used to reduce the fringe data either to be "hard coded" in the analysis program or to be applied by the operator during the analysis process. The rules may be simple ("For this class of photographs, start fringe numbering at the far left"), numerically complex ("Trace the fringe through the discontinuity by working backward from both sides of the discontinuity") or judgemental ("That is not a fringe").

These rules are not the type of information suitable for encoding within the body of a general purpose analysis routine. Rather they represent the criteria used to select the method of analysis to be applied. Consequently, a general purpose FAS capable of automatically analyzing a broad class of fringe data, must be able to recognize when to apply the expert knowledge of a trained operator, and to independently use this knowledge to control the analysis process.

Recently work in Artificial Intelligence has shown that Expert or Rule-Based systems are well suited for encoding rules and heuristics in a flexible format suitable for controlling the image segmentation process.(11) A FAS Expert Decision Module (EDM) would provide similar capabilities for a rule-based control of the fringe process.

In many respects an EDM resembles a conventional, small expert system. However, there are a number of significant differences. A traditional expert system is designed to mimic the reasoning of a human expert in solving a problem. Typical components of an expert system include an operator interface, an inference engine, and an expert knowledge data base. Such systems typically operate by requesting information from the operator or data base and using this information for making a judgment which the operator then acts on.

An EDM, on the other hand, is an integral part of the fringe analysis control loop and must mimic both the reasoning and actions of trained operator. The EDM acts autonomously as the operator of the FAS to request existing information from the FAS knowledge base, to command that a new action be taken by some module, and to directly control the extraction of the fringe data from the fringe image. Like any other component of the FAS, it is modular, able to both receive a command from the FAS and to send control commands back. It is a specialist in its narrow knowledge domain rather than being an all encompassing "expert system." The FAS may support more than one EDM, each being an expert in its own domain. This allows an EDM to call on a consultant EDM if knowledge outside its specialization is required.

5.1 Expert System Development Language

A number of expert system development language and tools were evaluated. A detailed discussion of this work is presented in Appendix F. Three criteria dominated the language selection process:

- Since fringe analysis is very CPU intensive and potentially quite time consuming even without an expert system, the language chosen must not appreciably slow down the system.

- o Expert Decision Modules must be able to start up and exit from the system rapidly.

- o The language must allow for creating a rule based system.

As a result of a extenive review of available tools, and hands-on evaluation of some tools, OPS5 was selected as the best currently available language for building a rule based system.

OPS5 is a non-algorithmic inference engine developed in the mid-1970s at Carnegie Mellon University. It allows the programmer to encode a set of rules quickly, efficiently, and conscisely. As a result of the non-algorithmic nature of OPS5, the programmer need not worry about the flow of control within the execution of the program. All control problems are handled by the resident OPS5 interpreter, making the OPS5 program or Production System, as it is often called, readily expandable and much less difficult to modify than a program written in a conventional, algorithmic language.

An OPS5 program is composed of a series of independent rules, called Productions. The OPS5 inference engine continualy scans existing Working Memory Elements (WMEs) (where the rules are stored) to see if the current conditions match the rules stored there. If the Production's conditions match the contents of the WMEs, the rule is said to "fire" and the Production's actions are performed and the contents of the WMEs updated. The process by which this occurs is refered to as the "Recognize-Act Cycle."

A typical OPS5 Production includes a Production Name, a number of conditions to match, and right arrow, and a number of actions to take if the conditions are met. An example of a generic OPS5 rule is:

```
(P Production-Name
       (Condition_1)
       (Condition_2)
       (.  .  .  .)
       (Condition_n)
--->
       (Action_1)
       (Action_2)
       (.  .  .  .)
       )
```

In this example, if and when the conditions Condition_1 to Condition_n all match the contents of the working memory, the production will "fire" and the actions specified by Action_1...Action_n will be performed.

5.2 Evaluation Of A Rule-Based Approach To Fringe Analysis

Once fringe processing modules have located candidate fringe contours and have represented them as line segments, the segments must be joined together or extended to form complete contours, mislocated segments must be removed, and the contours must be numbered in the correct order. This process represents the most critical, operator-intensive and error-prone step in the fringe

analysis process. This process is complicated by the fact that fringe contours may not be directly trackable across shock boundaries.

To address the problem of fringe ordering expert knowledge must be applied to the process either interactively by the operator or by the fringe analysis software. Current fringe analysis software such as Becker's (8) builds some of this knowledge into the analysis code, but significant operator input is still required.

An alternate approach is to create a more extensive set of expert rules for ordering fringes contour segments throughout the image. For example, in adressing the more general image segmentation problem, Nazif and Levine (11) developed a set of rules to control the processing of data representing line segments. These rules provide them with the ability to determine whether or not line segments detected in an image should be merged together or deleted. Using these rules, they first detect lines and edges in a digitized image and then determine whether or not these lines have missing segments.

Conventional fringe analysis algorithms also must address locating and recognizing existing fringes segments which are not apparent in the digitization process but which the human eye detects quite clearly.

Because of the direct bearing of Nazif and Levine's rules to fringe analysis, a subset of these rules was implemented in OPS5 to investigate the utility of applying expert rules to the fringe analysis process. For example, the following rule (number 1701) is used to merge a short line segment into a larger line segment.

```
RULE (1701)
  IF: (1) The FRINGE LENGTH is NOT LOW
      (2) The LENGTH of the FRINGE IN FRONT is LOW
      (3) The FRINGES are TOUCHING
      (4) The closest POINT IN FRONT is LOW

THEN: (1) MERGE the LINES FORWARD
```

The corresponding OPS5 production is:

```
(P Nazif_and_Levine_#1701

   ; this first clause chooses two fringes, one in front of the
   ; other, which satisfy clause (4) above, that is, the closest
   ; point on the fringe in front is LOW.
   (In_Front_Of ^Fringe_in_front_ID <fringe_in_front>
                ^Fringe_ID <fringe>
          ^Pos_of_Close_Point_in_Front << VERY_LOW  LOW >>)
   ; this second clause insures that the length of the fringe in
   ; front is LOW
   (Fringe ^Fringe_ID <fringe_in_front>
          ^Length << VERY_LOW  LOW >>)
```

```
    ; the third clause insures that the length of the fringe in
    ; question is NOT LOW
    (Fringe ^Fringe_ID <fringe>
            ^Length << MEDIUM  HIGH  VERY_HIGH >>)
-->

    ; if all the above clauses are satisfied, an external routine
    ; called Merge is called which will merge the two fringes
    (call Merge <fringe> <fringe_in_front> Forward)
    )
```

A number of rules similar to the above were implemented. The rule based system was then evaluated using using a wide range of coordinate data. The results showed that given a collection of rules suitable for fringe processing, OPS5 can easily be used to generate and efficient rule based system to assist in the fringe segmentation and numbering process.

5.3 Expert Decision Module Design Requirements

The overall architecture for the FAS is driven by the requirements for building expertise into the system, namely:

- The knowledge required to control fringe processing is dynamic. At any given processing step, existing information about the fringe image may change forcing previous decisions to be reconsidered.
- An Expert Decision Module must be able to independently request specific analysis steps and gather new knowledge from that analysis step.
- The FAS must not require the existence of expert knowledge for any given function but must be able to use such knowledge to assist fringe analysis if it is available.

To meet these goals, the FAS processing modules operate on the fringe data in discrete steps. Although an EDM is also a processing module, besides having access to the global Named Knowledge, it also has expert data consisting of rules, heuristics and meta-rules (how to apply rules) on how to process fringe images. An EDM controls the processing modules via Shell Callback using the Fringe Processing Command Language (FPCL). Because the FPCL is syntactically rigorous and consistent in command format, processing commands can be dynamically composed ("on the fly") by an EDM. This provides the EDM far greater flexibility for controlling processing than could be accomplished by embedding explicit commands within a control program or analysis script. Following the completion of a processing command, an EDM can examine the results of the action and select the next action to perform.

Implementing each EDM as an independent processing module has a number of significant advantages over more conventional expert system approaches. First, the EDM is to be small, efficient, and primarily a rule-based decision maker, using its knowledge to direct the activities of other processing modules. This division of activities allows CPU intensive fringe and image processing work to be performed in appropriate computational languages like Fortran yet still allows using an appropriate language (OPS5) for developing the rule-based

system.

Second, the limited scope of each EDM simplifies the development of the expert system rule-base by helping to minimize unexpected side effects caused by adding new rules to an existing system. On large rule-based systems with many complicated rules, great care must be taken that adding a new rule will not affect the operation of other rules and cause unwanted side effects. By keeping the rule-base of each EDM small and relevant to the problem it is addressing, side effects will be minimized and the EDM will be easier to develop and maintain.

Third, by limiting the scope of specialization for each EDM, a processing environment becomes possible in which the system can be easily augmented with new expert knowledge in incremental steps as needed. For example, if an EDM requires information or processing to be performed not within its area of expertise, it requests the assistance of a consultant EDM. From the viewpoint of the FAS monitor, the consultant EDM is just another EDM which has taken over control and is sending back commands for various processing modules to perform. When the consultant finishes performing its task, it exits, leaving behind for the initial EDM to use, the knowledge it has acquired and the processing it has had performed.

Establishing Requirements for a Fringe Analysis System

While the primary goal of this project was to design a software architecture for a "state-of-the-art", modular, digital Fringe Analysis System, it was also necessary to insure that the system KMS would propose would be appropriately focused and have the flexibility to meet the anticipated requirements for both NASA and private industry.

Consequently, during the Phase I study, Dr. Charles Vest evaluated the current requirements of NASA and private industry for a general-purpose, holographic fringe processing system.

Dr. Vest's study concluded that:

- The primary area for initial FAS development work to address is the aerodynamics field. Once a fully functional FAS has been developed, applications to HNDT (Holographic Non-Destructive Testing) may open up. Until that time, numerous applications exist in the aerodynamics field which would benefit immediately by the development of a FAS.

- Data analysis for the projects at the NASA Langley Cryogenic Wind Tunnel and at NASA Ames (among others) would benefit by the development of a FAS.

- For holographic fringe analysis to realize its potential, significant new research must be done to develop fringe location software and to automate the data reduction process as much as possible.

- There currently is no commercially-available fringe analysis package or equipment which is fully suitable for the applications at NASA Langley or Ames. Moreover, although great strides have been made in recent months fringe-finding algorithms, no general-purpose systems exist which have the capability of being adapted to a wide variety of problems.

- The complexity of the fringe analysis problem may lend it self to use of a rule-based or expert system.

Evaluation of Fringe Analysis Algorithms

To gain experience in fringe analysis problems typically encountered by aerodynamicists, an existing KMS fringe contour location package was used to analyze holographic fringe images acquired during both NASA Langley wind tunnel tests and an Ames Rotocraft experiments.

The KMS fringe analysis package is designed to work with fringe images acquired in laser-plasma interaction experiments which have:

- Simple, regular shapes.
- Monotonically increasing spatial frequency and decreasing fringe curvature.
- Low signal to noise ratio with the spatial frequency of the noise close to that of the fringe spacing.
- High granularity.

The package is able to routinely locate fringe contours embedded in noisy, low-contrast fringe data. Its approach is to linearly transform the coordinates of each curved fringe into a nearly vertical straight line, perform a sliding window row average to enhance the signal to noise ratio, locate the peak coordinates in each row for that fringe, and then transform the peak loci back into the initial fringe coordinate system.

To accomplish this, an approximate piece-wise-linear "guess" or first approximation is made for the shape of a fringe as is seen in Figure 4. This first approximation may be input from a stored template, calculated, or input interactively by an operator.

A linear transformation (shift) array is then constructed to transform the initial approximation fringe into a straight vertical line. This transformation is applied to each row of the digitally stored image and the signal to noise ratio of the target fringe, which is now approximately vertical, is enhanced by row averaging. The resulting peak coordinates representing points along the fringe center are converted back into the initial coordinate system by inverting the transform.
The existing fringe analysis package proved to be successful at semi-automatically extracting many fringe contours from wind tunnel inteferograms. Figure 5 shows the results of using this package to trace fringe contours obtained from two different sources. In Figure 5a, the flow field contours of a conical test object in a NASA Langley wind tunnel were located. Note that the fringe contours are correctly followed across the shock boundary. In Figure 5b, even the high density flow field contours generated at the Ames Rotocraft experiment were located.

While successful with this fringe data, the package still needs considerable improvement before it can be used with many different types of fringe data. While the analysis package effectively performs the functions for which it was designed, it is an example of using a technique based on specific knowledge of the field being examined - namely that the the basic topological structure and orientation of the fringe patterns produced in the plasma experiments is known.

Moreover, use of the package still requires considerable operator interaction particularly for analyzing strongly curved, complex fringe data or tracking fringes through a shock. It is easy for the software to get confused at a shock boundary and mistrack a fringe contour particularly if the discontinuity is sharp. The package's limitations under certain conditions suggest a number of areas which warrant further development, namely:

- Methods for locating and identifying shock boundaries.
- Methods for correctly matching up fringes across shock boundary layers.
- Algorithms for tracking and extrapolating high density fringe contours through a boundary layer.
- Algorithms for locating highly curved (circular, closed) fringes.
- Methods for decreasing operator interaction for analyzing complex fringe data.

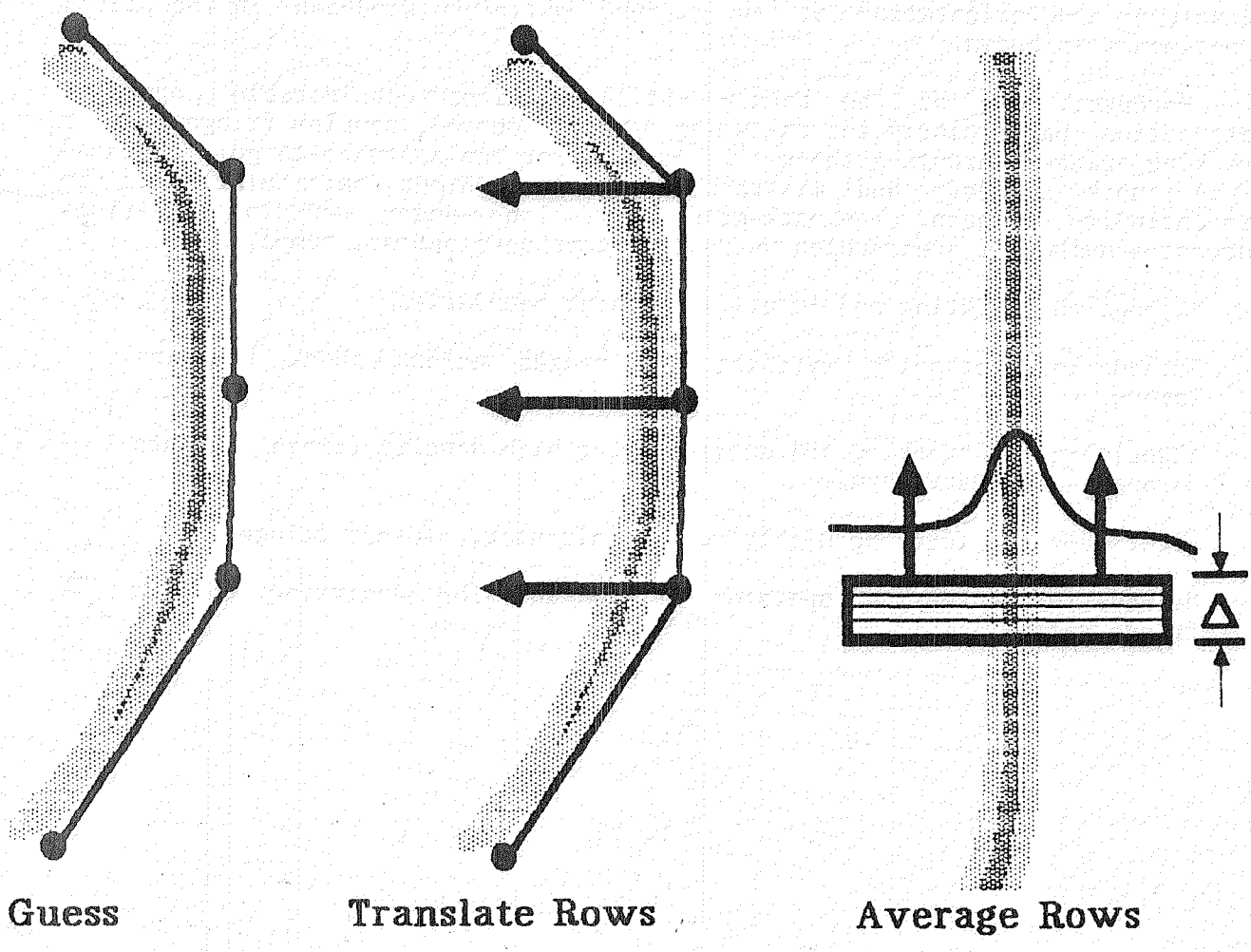
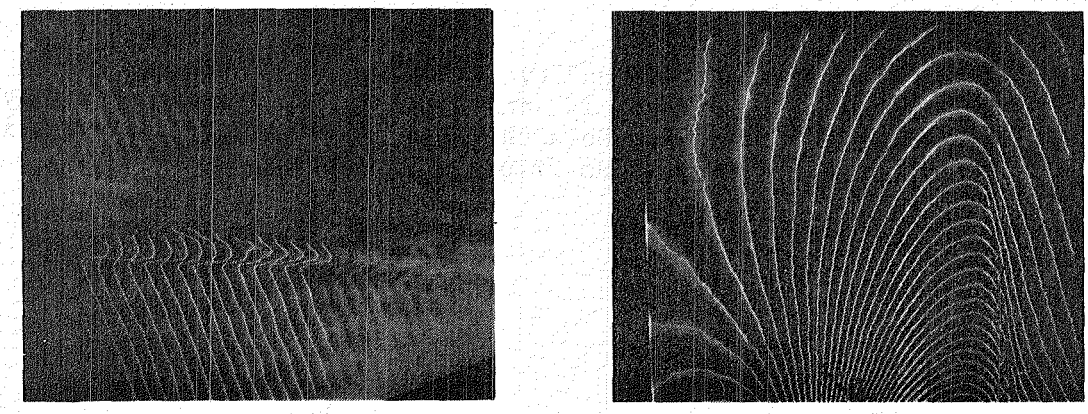


Figure 4. Using fringe straightening to enhance fringe signal to noise ratio.



(a) (b)

Figure 5. Semi-automatic Fringe Contour Location.
(Bright line are fringe contours found by the fringe analysis package overlaying the digitized fringe photographs)
(a) Langley Conical Test object
(b) Ames Rotocraft experiment

Validation Studies of The Proposed FAS Architecture

As the architectural design of the FAS evolved, the conceptual design was converted to software and the functionality of the implementation evaluated. Based on the results of the evaluations, the design was modified or refined. Concept testing and evaluation was performed in the following areas.

- o A prototype FAS Monitor was developed supporting the majority of its final design goals with the exception of Shell-Callback.
- o A prototype fringe processing language supporting five external and three internal commands was tested.
- o Five fringe processing modules were developed to evaluate the problems involved in creating a device independent Fringe Processing Command Language.
- o Subroutines were developed to provide ready access to the Analysis Shell's Named Knowledge elements.
- o NASA supplied photographs of fringe data were analyzed to demonstrate the suitability of our Fringe analysis algorithms for analyzing holographic wind tunnel fringe data.

- o OPS5 programs and interface subroutines were developed to demonstrate the feasibility of using OPS5 for writing Expert Decision Modules.

- o A simple Fringe Analysis Advisor was developed in OPS5 to evaluate the difficulty of using a rule based approach for developing a fringe locator EDM.

The intensive "design-evaluate-redesign" cycle applied to all parts of the FAS, has led to an architecture that is demonstrably applicable to analyzing holographic fringe data. Moreover, the architecture's flexibility potentially lends itself to applications other than fringe analysis.

References

1. G. Lee, Application of Holography to Flow Visualization, NASA Technical Memorandum 84325 (1984).
2. A. W. Burner and W. K. Goad, Combined Single-Pulse Holography and Time-Resolved Laser Schlieren for Flow Visualization, NASA Technical Memorandum 83109 (1981).
3. W. K. Goad and A. W. Burner, Holographic Flow Visualization at the Langley Expansion Tube, NASA Technical Memorandum 83116 (1981).
4. G. Lee, D. Buell, J. Licursi, and J. Craig, AIAA Journal **22**, 504 (1984).
5. L. T. Clark et al.,J. Fluid Eng. **99**,737 (1977).
6. R. W. Menzel and L. D. Vandergriff, Program for Automated Holographic Data Reduction, AEDC-TR-80-39, Aug. 1981.
7. F. Becker and Yung H. Yu, Opt. Eng. 429 (1985).
8. F. Becker, G. Meier, and H. Wegner, SPIE Appl. of Digital Image Processing IV 359,386 (1982).
9. F. Becker, G. Meier, and H. Wegner, Development of an Instrument for Evaluation of Interferograms, NASA Technical Memorandum NASA TM-76673 (1981).
10. J. Kittleson, A Holographic Interferometry Technique for Measuring Transonic Flow Near a Rotor Blade, NASA Technical Memorandum 84405 (1984).
11. A. M. Nazif and M. D. Levine, IEEE Trans. on Pattern Analysis and Machine Intelligence **PAMI-6**(5), Sept. 1984.
12. W. R. Funnell, App. Opt., **22**, 3245 (1981).

APPENDIX A

FUNCTIONAL DESCRIPTION OF ANALYSIS SHELL OPERATION

The phrase "Analysis Shell" is used to describe the analysis environment for two reasons. First, the FAS Monitor program invoked by the operator serves as a shell which holds both the processing modules and the global knowledge which any module may access. Second, the processing takes place in a series of nested processing levels or shells. Level 0 is the operator or script file input level.

The FAS Monitor translates Level 0 input into a command which is passed to a processing module in a Level 1 subprocess. Commands generated by active Level N module are passed back to the FAS Monitor (Shell-Callback) which sends the command back for processing in a Level N+1 subprocess. If the Level N+1 subprocess does not yet exist it will be created and initialized prior to dispatching the command to it. Shell-Callback can proceed up to the maximum nesting of processing levels allowed (an installation dependent parameter).

The FAS Monitor program performs the following functions when activated by an operator command.

- o Creates the Analysis Shell environment
 - Creates the FAS-specific job logical name table
 - Creates the primary subprocess and initializes it
 - Executes a FAS initialization file if requested
 - Executes any command-line specified processing script
 - Prompts the operator for a command
- o Provides for operator control of the FAS
- o Serves as the central command/communication dispatcher
 - Parses and sends operator commands to the appropriate internal subroutines, external FAS modules or DCL
 - Parses and sends script file commands to the appropriate internal subroutines, external FAS modules or DCL
 - Receives command requests from a processing module or EDM for the services of another module and retransmits that command to the appropriate module. Returns control to the original module when the secondary processing module terminates

- o Maintains and manages the subprocesses within which the FAS processing and EDM modules run.

- o Manages the process job logical name table in which the "Named Knowledge" data is stored.

- o Manages script files.

 - Logs operator commands to script files
 - Executes command input from script files
 - Conditionally executes script commands
 - Branches to specific sections of the script file

In Figure A1, the interaction between the FAS monitor program, the operator, a Shell initialization script, an analysis control script file, a logging file, an EDM, a processing module and Shell Callback is schematically shown. In this particular example, the operator started the monitor program which created the Analysis Shell, invoked an initialization file and then started taking commands from the script file specified by the operator who started the FAS Monitor program. The script file, turned on logging (to track what commands the Expert Decision Module would select) and then invoked the EDM which started processing the fringe image to locate fringe contours. At some point, the EDM invoked the PM2 processing module, and in the figure is waiting for PM2 to complete its processing and update the Named Knowledge data base. Also in seen in the figure is an empty subprocess which is available to take a Shell Callback command from the PM2 processing module if necessary.

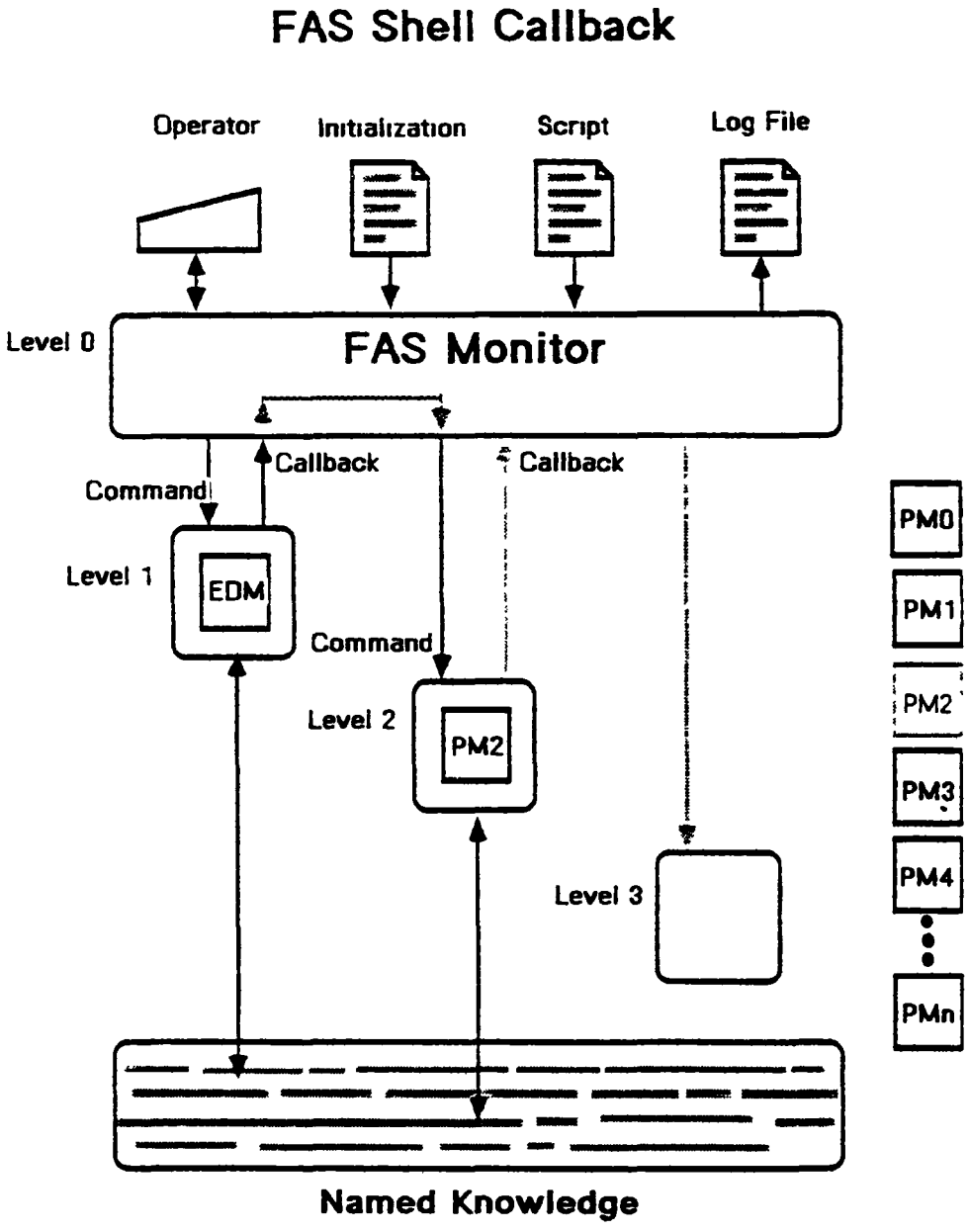


Figure A1. The FAS in Action

APPENDIX B

DETAILED DESCRIPTION OF THE FAS MONITOR PROGRAM

NOTE

The following sections assume the reader is familiar with VAX/VMS software terminology. Their inclusion in this report serves to document the detailed software design work accomplished during the contract period. Unless other wise noted, the FAS Analysis Shell as described herein, has been prototyped and tested for functionality.

B.1 Creation And Initialization Of The Analysis Shell Environment

Assuming the FAS software has been installed on a VAX system correctly, an operator wishing to use the FAS software logs on to the VAX host computer, and enters the command

```
$FAS  [Script_File]
```

Normally, the FAS command is defined to be

```
FAS:==$FAS$LIBRARY:FAS_Monitor
```

Once the FAS monitor is invoked the following initialization steps are performed.

1. If a script file is specified on the command line, the file name is saved so that the script file may be invoked as soon as all initialization is complete. This facility is useful when reducing numerous fringes of the same type whose analysis can be expected to follow along relatively similar lines. The name of the script file may be any legal VMS File name. However, if the file type is omitted, the file type .FAS is assumed **(Designed but not yet implemented).**

2. The FAS Monitor creates the FAS Job Logical Name Table. The maximum size this table is controlled by the operator's account profile. The size of the table needed depends on the complexity of the analysis problem. **(Partially implemented)**

3. The FAS Monitor creates the primary subprocess for communication with the Level 1 processing modules and establishes an exit handler to insure that the subprocess does not vanish without notifying the the FAS Monitor.

4. Basic initialization of the primary subprocess is performed to a) establish the terminal as the input device, b) disable extraneous error messages, and c) establish the FAS command to the subprocess DCL command table.

5. If the Logical Name FAS$INITIALIZE is defined, the FAS Monitor invokes this name as an initialization script file. This script file can be used to load the knowledge base with information which is common for all the fringe analysis tasks to be performed (Designed but not yet implemented).

6. If a script file was specified on the initial FAS command line, it is invoked (Designed but not yet implemented).

7. When processing of script files is complete, the operator is prompted for a interactive command with,

FAS>

B.1.1 Operator Control Of The FAS

When the FAS> prompt appears on the operators terminal, the operator can enter four types of commands, namely

1. An internal command. The input is first checked to see if it exists in the internal command table. Internal commands are handled within the FAS Monitor program itself. The internal commands supported include:

 1. EXIT. If EXIT is entered in response to a FAS> prompt, or read from a script file, all open files are closed, all analysis ceases, and the FAS Monitor, exits returning the user to the VAX/VMS DCL level.

 2. LOG <file-spec>. When the LOG command is used all operator commands entered to the FAS> prompt will be logged to the file specified by the <file-spec>. If the <file-spec> is omitted, the log file defaults to FAS_COMMAND.FAS. If the file type is omitted, the file type defaults to .FAS. Operator commands will be logged to the file until the operator enters a NOLOG command.

 3. NOLOG. The NOLOG command turns command logging off and closes the open log file.

 4. HELP. The HELP command accesses the FAS help file which provides the operator with help on using the FAS commands.

2. A command to open a script file for processing. If the first character of any input stream is an '@', it is assumed that all following characters are the name of a script file. An attempt is then made to open a script file with that name and if successful the script file is read in a line at a

time, and each command line is processed as if it were an operator input. Script files also can reference script files up to 8 levels deep.

3. A command to send to the operating system. Any command preceeded with a '$' is assumed to be a DCL command and is sent directly to the DCL CLI (command line interpreter) for processing in the appropriate level subprocess. The exit status of each DCL command is checked, and if the status is not success, the FAS will issue an appropriate error message.

4. All other commands are assumed to be valid FAS commands. These commands are internally prefixed with a "FAS/" and sent to the appropriate subprocess level where they will be parsed by the VMS CLI routines, and the required processing module will be activated.

B.1.2 Subprocess Control And FAS Monitor Communication

Two types of command communication channels exist between the FAS Monitor and the processing modules. The first is the command channel. which is a bidirectional channel between the operator terminal and main process and the subprocess. This channel is set up using the FAS Subprocess control subroutine package which provides three basic subroutine functions

```
SUB_CREATE -- Create a subprocess
SUB_SEND   -- Send a command line to the subprocess
SUB_END    -- Delete a subprocess
```

When a subprocess is created, a mailbox is established as SYS$INPUT for that subprocess. From this point on, the copy of DCL running in the subprocess will take its commands from the mailbox. The SUB_CREATE subroutine returns a pointer, so that the SUB_SEND and SUB_END routines can direct their commands to the correct subprocess. As soon as the subprocess is established and an exit handler for it established, the subprocess is initialized by 1) the FAS command to be a valid CLI command, and 2) setting the SYS$INPUT to be identical to the SYS$OUTPUT device (the terminal) for all other images running in the subprocess (but DCL still takes its commands from the Mailbox).

The second communication channel is the Shell-Callback mailbox (**Designed but not yet implemented**). When a Level N processing module requests additional concurrent processing of an additional FAS command, it writes the command to the Shell-Callback mailbox and hibernates. When the command is written, the FAS Monitor is notified via an AST routine, reads in the command requested from the Level N module and dispatches the command to a Level N+1 subprocess. When the Level N+1 subprocess completes, the FAS Monitor then wakes the Level N subprocess.

B.2 Use Of Job Logical Names Tables

The FAS knowledge base is stored both as Named Knowledge in the FAS Job Logical Name Table and as ancillary data and image files. Knowledge may be placed into the logical name table by executing a DEFINE command in the specified subprocess or using the Put_Named_Knowledge subroutine (**Designed but not yet implemented**). Knowledge may be read from the logical name table using the Get_Named_Knowledge

subroutine (Partially implemented).

B.3 Detailed FAS Monitor Logic Flow

In order to understand the operation of the FAS, it is necessary to examine in detail a number of its capabilities and the logic flow of a command as it passes through the system.

B.3.1 FAS Command Processing

The FAS can take commands from three sources, the operator, a script file, or a processing module via ShellCallback. The processing of a command is identical regardless of the command source. First the command line is normalized to a standard format. Leading and trailing spaces or tabs are removed. All characters are converted to upper case and multiple spaces converted to a single space except for strings enclosed in quotes (Partially implemented)

Each input command line is checked to see what type of command it contains. The FAS Monitor checks to see if the command is

1. An internal command. Internal commands are contained in a CLD file SHELLCMD.CLD. The CLD file is compiled and linked with the FAS monitor program. The SHELLCMD.CLD file specifies the action routine to automatically invoke if a command is present on the command line.

2. A command to open a script file for input. If an '@' sign is encountered, the rest of the command line is taken to be a VMS file specifier and the Shell attempts to open a file of that name to use for command input.

3. A DCL command. If a leading '$' is encountered, the rest of the command line is assumed to be a VMS command and is sent to DCL for processing.

4. A FAS command. Any other legal input will be assumed to be a valid FAS command. Illegal input will generate an error message.

After each command is processed by a subprocess, the Shell will check the exit status of the module processing the command and will display an error message if the status is not successful. It will do this by sending a command to that subprocess to place the modules exit status in the Job Logical Name table where the shell can read it.

B.3.2 Direct Operator Command

Any time the

FAS>

prompt is present, the operator can enter a command which will be parsed and dispatched appropriately. Whenever, the operator is prompted with FAS>, it means

that processing has stopped and that the shell is at Level 0.

B.3.3 Script File Processing

Whenever a leading "@" sign is encountered in a command line, the remainder of the command line is considered to be the name of a a VMS script file. If the script file exists, the Shell opens the file, reads in the commands a line at a time, normalizes the input lines, parses the command line, and executes the command line exactly as if it had been input by an operator.

Command scripts may be nested up to 8 levels deep but commands are taken only from the most deeply nested script file until that script file is closed or a more deeply nested script file is opened.

Normally the contents of the script files are commands to be sent to external modules. However, script file processing supports three commands which are used to control the script processing itself. These are

- Labels within the script file. **(Designed but not implemented)**
- The GOTO command. **(Designed but not implemented)**
- The IF command. **(Designed but not implemented)**

The GOTO and the label are designed so that it is possible to branch from one section of the script file to the label specified with the GOTO command. When a GOTO <label> command is encountered, the script file will be repositioned to its start and read in a line at a time searching for the label. When the label is found, script file processing will resume at that point. If the label is not found, processing will terminate and an error message will be displayed.

The IF command will allow for conditional branching and conditional execution of script file commands based on matching criteria in the Shell global data base.

B.3.4 Shell Call-Back

NOTE

Shell-Callback is designed but not yet implemented

After each subprocess main communication channel is established, a Shell-Callback mailbox communication channel between the main process and each subprocess will also be established and the Shell will establish a write attention AST for it.

When a processing module wishes concurrent processing to be done by another module, it will write the command to be processed to the Callback mailbox and hibernate. The Shell will be notified of the write by the write attention AST which will set a flag to show that a Callback command is incoming (Callback mailbox full

flag), and wake the main process. The main process will again get ready for the next input command but because the Callback flag is set will read the command from the Callback mailbox.

Prior to issuing the command to a subprocess, the Shell will increment a Callback level counter to show at what depth callback commands are being processed, clear the Callback Mailbox full flag and then send off the command to the next level deeper subprocess (and hibernate until awakened by an AST). When the main process again is woken up, it will check to see if the Mailbox-full flag is set. If it is not set, the Callback command completed so the Call back level counter will be decremented and the proper subprocess notified (by waking it up) that the command completed. If it is set, another callback command will be processed.

APPENDIX C

FAS INTER-OBJECT COMMUNICATION ARCHITECTURE

Because the FAS is implemented as a collection of independent software modules, it is inherently very flexible. This flexibility is forged into a coherent analysis package by imposing a common communication architecture onto the modules in the system.

The Communication Architecture views the FAS as a set of objects each of which is able to perform four generic Analysis Shell functions in addition to object-specific analysis functions. The objects supported by the Analysis Shell are:

- o The FAS operator.
- o Shell Script Files.
- o Fringe processing modules.
- o Expert Decision modules.

While the analysis functions to be performed are primarily in the image analysis and fringe analysis domains, the FAS architecture allows any type of analysis to be performed. As seen in Figure C1, each object may either send or receive information to or from the Analysis Shell. This information may either be commands directing the next action to take or fringe knowledge generated by or needed for the numerical processing of the fringe data. Specifically, each object can:

- o receive an action command and take the requested action.
- o request knowledge by name from the FAS global knowledge base.
- o create or modify knowledge which it then places in the FAS global knowledge base.
- o send/relay additional commands through the Monitor for additional actions to by performed another (but unknown) object.

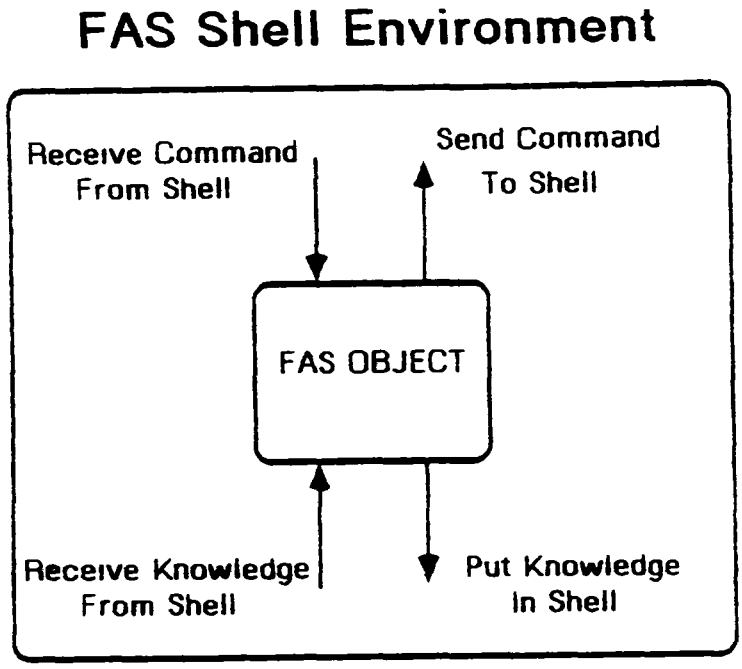


Figure C1. The Generic FAS Object

The FAS Monitor's primary function is the communication of information (action commands and fringe knowledge) between the various objects in the FAS. To do this the FAS Monitor establishes a processing environment in which it creates and maintains communication channels between itself and the processing modules.

Communication between the objects is facilitated by the common Fringe Processing Command Language shared between them. Whether a command is input by an operator, read from a script file, or passed back to the FAS Monitor from an EDM, the command format is identical. Moreover, consistent command syntax rules and device independence, allow an EDM to construct a command "on-the-fly" without having to consider a wide variety of special cases.

Figure C2 schematically shows four FAS objects with active communication channels. In this figure, two features should be noted. First, while each object has an "open" and "active" communication channel, only the last object in the chain is executing code. Second, since each object has independent access to all elements of the FAS Knowledge base, if module A activates module B, module A must assume that any or all elements of the knowledge base may have been modified while module B was active.

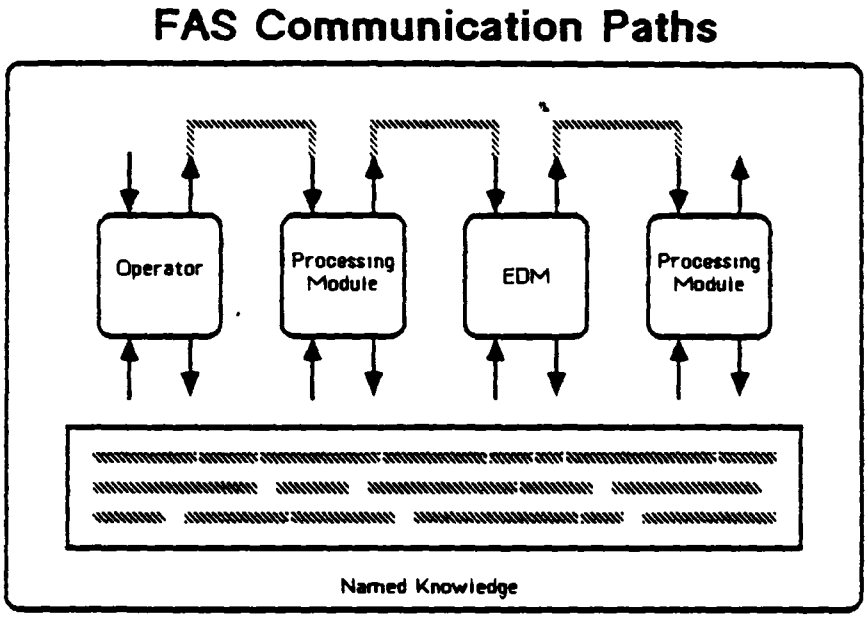


Figure C2. FAS Object Communication Paths

APPENDIX D

NAMED KNOWLEDGE ARCHITECTURE AND VMS LOGICAL NAMES

D.1 Establishing Knowledge Naming Conventions

During the course of development of the FAS, a lexicon of names for the knowledge created and requested by modules will be developed. By knowing the name specified for a given piece of information, any module may request or update that information in an unambiguous manner.

While it will be possible to define "Alias" names to point to information specified by another name, the initial FAS development work will not attempt to address the numerous problems inherent in knowledge representation ambiguities which current AI research into Natural Languages addresses.

For example, while the English language allows essentially equivalent knowledge to be transferred in a variety of ways, the interpretation of the transferred information depends on significant amounts of knowledge external to the information itself.

As a case in point, consider a simple request such as "How many fringes are present". Valid answers could include "24", two dozen, or "Too Many". Of these, answers the first is numeric, the second is numeric but requires a conversion to numeric format, and the third is a totally fuzzy concept which would require all FAS modules to know what the definition of "Too Many" is.

Instead, the Named Knowledge passed between the processing modules will be encoded into pre-defined, unambiguous data types. If a knowledge element is to be a numeric quantity, only a numeric quantity will be allowed to be encoded into a numeric knowledge element.

D.2 Named Knowledge Data Types

Named Knowledge is data (knowledge) of varying types encoded as ASCII text strings and stored by name. The text strings may store arbitrary data including names of additional Named Knowledge data elements. The VAX/VMS implementation of the FAS Architecture will store this data in the VMS Job Logical Name table. However, a non-VMS implementation of this architecture could provide similar functionality via common areas and linked lists.

To provide these capabilities each piece of named knowledge has three attributes.

- o The name by which the information may be retrieved.
- o The data type of the knowledge so that the ASCII text may be translated into the proper data format by a routine which knows only the name for the knowledge.
- o The knowledge itself which is encoded as an ASCII text string.

Each Named Knowledge element name is composed of two parts; the type designator and the data name. Together, they form a Named Knowledge element which can be translated into an equivalence text string in which the data is stored. The type designator specifies the format that the text data is to be translated into (text, a real number, an integer, etc.). Using simple character tests, a subroutine can rapidly decide which data type the name represents and translate the data appropriately. For example:

FAS__F__name	VAX/VMS filename
FAS__I__name	Integer data
FAS__L__name	Logical data (True/False)
FAS__P__name	Knowledge element pointer
FAS__R__name	Real number data
FAS__T__name	Textual data

D.3 Named Knowledge States

When a module requests a named knowledge element two responses can occur. Either the information exists and the Shell returns the current information to the module, or the information does not exist at all and the Shell notifies the module that the information does not exist.

If the information does not exist, the module may "know" how such information might be obtained. For example, it might pass a request back through the Shell to an EDM to go find that piece of knowledge.

If the knowledge can be found, the module is notified that it is available and continues. If the requested knowledge can not be found, and error message will explain why and the returned Shell Call-back status will inform the requesting module that the information still can not be obtained. If the unavailable knowledge is required for continued processing, the module will halt, display the name of the unavailable Named Knowledge element, and provide an opportunity for the programmer/operator to investigate why the knowledge is not available.

D.4 VMS Logical Names

Each process on a VAX/VMS system can create a Logical Name and define it to be some arbitrary text string. The Logical Name and the text string associated with it may each be up to 255 alpha-numeric characters long. If the logical name is created

in the Job Logical Name table, both the main VMS subprocess (the Shell) and each subprocess (the processing modules) can access a logical name and request that it be translated into its defined text string.

In addition to storing simple text strings, logical names can themselves store logical names (just another text string) and any program can request that the additional logical name also be translated into its's equivalent text string.

APPENDIX E

FRINGE PROCESSING COMMAND LANGUAGE ARCHITECTURE

Within the FAS, images are manipulated by using the Fringe Processing Command Language. The language, and associated processing modules, are designed to support device independence. Images may be input using technologies such as CCD, photo-diode or video digitizers. Images may be on disk or tape file, or reside in VAX memory. Images may be displayed on a simple frame buffer or a complex image processor. Device independence makes it easy to re-configure hardware for a particular application. Devices may be switched by simply incorporating new device driver routines.

The Processing Module Command Language uses the following format:

[$FAS/]COMMAND/qualifier_1,...,/qualifier_m parameter_1...parameter_n

If the command is issued from the VMS DCL level, it must be preceeded by **$FAS/** and the **FAS** command must have been established for the user's process. If the command is to be entered to the Shell prompt (FAS>), or is embedded in a FAS script file, the **$FAS/** prefix must be omitted.

The COMMAND specifies the FAS command to be executed. The qualifiers describe or modify the action taken by the command. The parameters specify what the command acts upon. The VAX/VMS CLI utility subroutines are used to parse and interpret the command, qualifiers and parameters.

The Processing Module Commands are divided into 15 groups:

- Auxiliary image information
- Complex filters
- Display control
- Edge detection
- Feature identification
- Geometric transformations
- Image combination
- Image input/output control
- Image statistics
- Neighborhood operations
- Noise reduction
- Pixel transfer functions
- Region of interest
- Template generation

o Transforms

The following conventions are used in specifying the processing module commands.

[] - Square brackets indicate that the enclosed item is optional.

<> - Angle brackets indicate that the enclosed item is a single choice of several options.

| - Separate the choices.

AUXILIARY IMAGE INFORMATION

SCALE - Return the scale factor (pixels/inch) of an image.

```
$SCALE  <image> file_specification

     <image> = channel_number | file_specification
```

COMPLEX FILTERS

COMPLEX_FILTER - Apply a complex filter to an image.

```
$COMPLEX_FILTER [<roi>] <filter_type> <image>

            <roi> = /ROI=INSIDE | /ROI=OUTSIDE
    <filter_type> = /CONSTANT   | /CIRCLE | /SINUSOID | /GAUSSIAN |
                    /HANNING    | /BARTLETT
          <image> = file_specification | channel_number
```

DISPLAY CONTROL

CLEAR - Clear (zero) an image or overlay.

```
$CLEAR [<roi>] <image>

      <roi> = /ROI=INSIDE    | /ROI=OUTSIDE
    <image> = channel_number | file_specification | /OVERLAY
```

DISPLAY - Display a channel in either black & white or pseudo color. The pixels may be displayed with a continuous wedge or discrete steps.

```
$DISPLAY [<roi>] <type> <format> /LOW_LIMIT=z_1 /UPPER_LIMIT=z_2 Channel_Number

       <roi> = /ROI=INSIDE | /ROI=OUTSIDE
      <type> = /PSEUDO     | /GREY
```

```
<format> = /CONTINUOUS | /DISCRETE=number_of_divisions
```

INITIALIZE - Initialize the image display.

```
$INITIALIZE
```

TEXT - Write text onto the graphic overlay. The text may be included in the call, or may be obtained from the keyboard. The starting location may be obtained from the cursor, keyboard or included in the call.

```
$TEXT <text_string> <location>

      <location> = /CURSOR | /KEYBOARD | /COORDS=(x_1,y_1)
   <text_string> = /STRING="..." | /STRING=KEYBOARD |
                   /STRING=file_specification
```

VECTOR - Draw a vector into the graphic overlay. The pixel coordinates may be included in the call, or may be obtained from the cursor or from the keyboard.

```
$VECTOR <coordinates>

   <coordinates> = /CURSOR | /KEYBOARD | /COORDS=(x_1,y_1,x_2,y_2)
```

VIEW - View an image in a channel.

```
$VIEW channel_number
```

EDGE DETECTION

EDGE - Apply edge detection operators to an image.

```
$EDGE [<roi>] <detection_type> <image>

              <roi> = /ROI=INSIDE        | /ROI=OUTSIDE
   <detection_type> = /GRADIENT=POINT    | /GRADIENT=AREA[_MAXIMUM] |
                      /GRADIENT=PLUS_X   | /GRADIENT=PLUS_Y   |
                      /GRADIENT=MINUS_X  | /GRADIENT=MINUS_Y  |
                      /FILL_IN=SIMPLE    | /FILL_IN=ADAPTIVE | /CLOSE_CURVE
            <image> = channel_number     | file_specification
```

FEATURE IDENTIFICATION

DETECT - Detect different classes of objects in an image and put the locations in a file. The locations may be boundary edges, centroids or an object mask.

```
$DETECT [<roi>] <feature> <output> <image> file_specification

   <roi> = /ROI=INSIDE | /ROI=OUTSIDE
```

```
<feature> = /FRINGE | /OBJECT | /SHOCK
<output> = /BOUNDARY | /CENTROID | /MASK
<image> = channel_number | file_specification
```

GEOMETRIC TRANSFORMATIONS

GEOMETRY - Apply geometric transformations to a image.

```
$GEOMETRY [<roi>] <transform> <image>

      <roi> = /ROI=INSIDE | /ROI=OUTSIDE
<transform> = /SHIFT=(x_columns,y_rows)            |
              /MINIFY=AVERAGE  /FACTOR=factor      |
              /MINIFY=DECIMATE /FACTOR=factor      |
              /MAGNIFY=INTERPOLATE /FACTOR=factor  |
              /MAGNIFY=REPLICATE /FACTOR=factor    |
              /ROTATE=angle                        |
              /X_FLIP    | /Y_FLIP                 |
              /EXCHANGE  | /TRANSPOSE
    <image> = channel_number | file_specification
```

WARP - Apply a spatial transformation to an image. The control grid may be input from a file or interactively generated.

```
$WARP <control_input> <interpolation_method> <image>

       <control_input> = /INTERACTIVE       | /GRID=file_specification
<interpolation_method> = /NEAREST_NEIGHBOR  | /BILINEAR
               <image> = channel_number     | file_specification
```

IMAGE COMBINATION

ARITHMETIC - Apply arithmetic operations to images or constants and put the result into an image. Underflow and overflow are set to 0 and 255 respectively. To prevent underflow or overflow, the input images may be scaled (divided by 2) when performing addition or subtraction.

```
$ARITHMETIC [<roi>] <operation> <image_1> <image_2> <output_image>

          <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
    <operation> = /ADD [=SCALED]  | /SUBTRACT [=SCALED] | /MULTIPLY | /DIVIDE
      <image_1> = channel_number  | file_specification  | constant
      <image_2> = channel_number  | file_specification  | constant
<output_image> = channel_number   | file_specification
```

LOGICAL - Apply logical operation to images or constants and put the result into an image.

```
$LOGICAL [<roi>] <operation> <image_1> <image_2> <output_image>

            <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
      <operation> = /AND | /OR | /XOR
        <image_1> = channel_number | file_specification | constant
        <image_2> = channel_number | file_specification | constant
   <output_image> = channel_number | file_specification
```

INPUT/OUTPUT CONTROL

CALIBRATE - Calibrate the digitizer for both bias and gain.

```
$CALIBRATE
```

COPY - Copy an image.

```
$COPY [<roi>] <input_image> <output_image>

            <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
    <input_image> = channel_number | file_specification
   <output_image> = channel_number | file_specification
```

CORRECT - Use the bias and gain values created by CALIBRATE to correct a digitized image.

```
$CORRECT <image>

        <image> = Channel_Number | File_specification
```

DIGITIZE - Digitize an image into a channel. Digitizer noise reduction may be performed by averaging a number (power of 2) images together.

```
$DIGITIZE [<roi>] [/AVERAGE=number] channel_number

     <roi> = /ROI=INSIDE | /ROI=OUTSIDE
```

IMAGE STATISTICS

PROFILE - Obtain the intensity profile of the line between two pixels and output it to a file. The pixel coordinates may be included in the call, or may be obtained from the cursor or from the keyboard. A line average may be specified and the profile may be displayed on the overlay plane.

```
$PROFILE <coordinates> [/AVERAGE=] [/DISPLAY] <image> file_specification

     <coordinates> = /CURSOR | /KEYBOARD | /COORDS=(x_1,y_1,x_2,y_2)
```

```
<image> = channel_number | file_specification
```

HISTOGRAM - Compute the histogram of an image and output it to a file. Optionally, the histogram may be displayed on the overlay plane.

```
$HISTOGRAM [<roi>] [/DISPLAY] <image> file_specification

          <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
<image>_input = channel_number | file_specification
```

STATISTICS - Find the min, max and compute the mean, mode and standard deviation of an image and output them to a file.

```
$STATISTICS [<roi>] <image> file_specification

    <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
<image> = channel_number | file_specification
```

NEIGHBORHOOD OPERATIONS

FILTER - Apply a filter to an image. The filter types are mean, gaussian, laplacian or arbitrary. If the filter type is /MEAN, the window dimensions may be included in the command or may be input from the keyboard.

If the filter type is /ARBITRARY, the kernel may be input from the keyboard or from a file.

```
$FILTER [<roi>] <operation> <image>

      <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
<operation> = /MEAN=KEYBOARD | /MEAN=(x_size,y_size) | /GAUSSIAN |
              /LAPLACIAN     | /ARBITRARY=KEYBOARD |
              /ARBITRARY=file_specification
    <image> = channel_number | file_specification
```

NOISE REDUCTION

NOISE_REDUCTION - Apply noise reduction operators to an image.

```
$NOISE_REDUCTION [<roi>] <reduction_type> <image>

            <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
<reduction_type> = /MODAL          | /ODD=DOT |
                   /ODD=LINE       | /MEDIAN
          <image> = channel_number | file_specification
```

PIXEL TRANSFER FUNCTIONS

EQUALIZE - Perform a histogram equalization on the image.

```
$EQUALIZE [<roi>] <image>

      <roi> = /ROI=INSIDE     | /ROI=OUTSIDE
    <image> = channel_number | file_specification
```

LINEAR_FUNCTION - Use an arbitrary piece-wise linear function to modify the lookup table or pixel values in an image.

```
$LINEAR_FUNCTION [<roi>] <modify> /FUNCTION=file_specification <image>

     <roi> = /ROI=INSIDE | /ROI=OUTSIDE
     <modify> = /MODIFY=IMAGE | /MODIFY=LUT
     <image> = channel_number | file_specification
```

NORMALIZE - Apply a contrast stretch to the image so that the lowest pixel value is 0 and the highest pixel value is 255. The stretch may be applied to an image or to the lookup table.

```
$NORMALIZE [<roi>] <modify> <image>

     <roi> = /ROI=INSIDE | /ROI=OUTSIDE
     <modify> = /MODIFY=IMAGE | /MODIFY=LUT
     <image> = channel_number | file_specification
```

POINT_CHANGE - Change the lookup table or image so that all pixels with value z_1 are changed to value z_2.

```
$POINT_CHANGE [<roi>] <modify> /IN=z_1 /OUT=z_2 <image>

     <roi> = /ROI=INSIDE | /ROI=OUTSIDE
     <modify> = /MODIFY=IMAGE | /MODIFY=LUT
     <image> = channel_number | file_specification
```

RANGE_CHANGE - Change all pixels in the range z_1..z_2 to the range z_3..z_4. The change may be made in the image or just the lookup table.

```
$RANGE_CHANGE [<roi>] <modify> /IN=(z_1,z_2) /OUT=(z_3,z_4) <image>

     <roi> = /ROI=INSIDE | /ROI=OUTSIDE
     <modify> = /MODIFY=IMAGE | /MODIFY=LUT
     <image> = channel_number | file_specification
```

THRESHOLD - Pixels >= z_1 [and <= z_2] are set to 255. All other pixels are set to 0. The change may be applied to the image or just to the lookup table.

```
$THRESHOLD [<roi>] <modify> /BOUND=(z_1[,z_2]) <image>

     <roi> = /ROI=INSIDE | /ROI=OUTSIDE
     <modify> = /MODIFY=IMAGE | /MODIFY=LUT
     <image> = channel_number | file_specification
```

REGION OF INTEREST

CREATE_ROI_BOUNDARY - Create a bounded region on an image. Subsequent processing may then be limited to that region. The region may be rectangular or arbitrary in shape. The boundary may be drawn in the overlay plane. The boundary coordinates may be obtained from the keyboard, cursor, a data file or a binary image.

```
$CREATE_ROI [/DISPLAY] <roi_type> <coordinate_input>

     <roi_type> = /RECTANGULAR <coordinate_input> |
                  /ARBITRARY <coordinate_input> |
                  /MASK <image>
     <coordinate_input> = /CURSOR | /COORDINATES=file_specification |
                          /KEYBOARD
     <image> = channel_number | file_specification
```

DELETE_ROI_BOUNDARY - Delete the region of interest boundary. i.e. process the whole image.

```
$DELETE_ROI__BOUNDARY
```

TEMPLATE GENERATION

TEMPLATE - Generate mathematical images.

```
$TEMPLATE [<roi>] <type> <image>

       <roi> = /ROI=INSIDE | /ROI=OUTSIDE
      <type> = /CONSTANT=n | /WHITE_NOISE |
               /GAUSSIAN   | /RAMP |
               /SINUSOID[=DAMPED]
     <image> = channel_number | file_specification
```

TRANSFORMS

TRANSFORM - Apply a standard transform [or an inverse transform] to an image. The transformed image is written to a file.

```
$TRANSFORM [<roi>] <transform> <image> file_specification
```

```
      <roi> = /ROI=INSIDE          | /ROI=OUTSIDE
<transform> = /HADAMARD[=INVERSE] | /FOURIER[=INVERSE]
    <image> = channel_number      | file_specification
```

APPENDIX F

IMPLEMENTATION CONSIDERATIONS FOR EXPERT DECISION MODULES

To insure that an EDM could be implemented which is compatible with the proposed FAS architecture, current software approaches to engineering rule based expert systems were investigated, several possible EDM implementation languages were evaluated and a simple EDM was implemented to test concepts.

The result of this work is an preliminary EDM functional design. The proposed design, specifies the implementation language to be OPS5, describes how the EDM will acquire knowledge from the analysis shell and control processing steps.

F.1 Selecting A Suitable EDM Implementation Software

It is anticipated that many fringe analysis decisions will have to be made on the basis of knowledge which is imprecisely known or which is not numerically quantifiable. Representing such knowledge is best done via textual identifiers. To engineer an EDM it is necessary to develop a rule based system which uses this knowledge to control and guide the analysis of fringe data. Since conventional procedural languages (eg, Fortran, Pascal, PL/I, etc.) are not particularly well suited for this application, we evaluated several alternate approaches for building EDMs.

The LISP, OPS5, and PROLOG AI languages as well as several expert system building tools were evaluated for use in developing an EDM. In addition, using a high level language for developing an EDM was considered.

Because an EDM is only a small part of the FAS, it is important to keep the EDM software development costs in perspective with the entire software package. Consequently, recently developed, and quite expensive, expert system building tools were ruled out. Likewise, VAX PROLOG was ruled out because it was quite expensive and not yet available for evaluation.

Developing an inference engine using a high level language was also briefly considered. After some study, it was felt that the labor costs of developing an inference engine in house would be quite large. Consequently, in-house development of an inference engine was also dropped from further consideration.

The remaining languages considered were LISP and OPS5 for developing the Expert Decision Modules.

F.1.1 Evaluation Of LISP

Two LISP packages for the VAX were evaluated, NIL and DEC's COMMON LISP. Both LISP implementations suffered from common failings. The LISP programs were large and took far too much memory. In addition, LISP applications were both slow (ponderous) to activate and to exit from the system, and seemed to require far too much in the way of CPU resources. As a consequence, the use of LISP for developing an EDM was dropped from serious consideration.

F.1.2 Evaluation Of OPS5

DEC VAX OPS5 was evaluated and found to be an excellent language to use in developing Expert Decision Modules. It is easy to use, and applications developed using it activate rapidly, quickly evaluate large rule bases, and are easy to interface to subroutines written in other VAX languages. In addition, OPS5 is a relatively inexpensive software product, and a very inexpensive run-time only license is available. This latter fact is important if the current research is develop into a cost-effective technology which can be marketed.

OPS5 is a relatively new language specifically designed for building expert or rule-based systems. Using OPS5, McDermot et. al. developed R5 which later evolved into XCON. XCON is used by DEC to configure VAX system and is widely considered to be the single most successful example of an expert system in daily use.

OPS5 is referred to as a "Production System". Each OPS5 program kernel automatically incorporates and "inference engine" interpreter which repeatedly executes a Recognize-Act cycle on all rules in working memory. When a match is found between a rule condition and the current working memory elements, the actions to be performed upon satisfying the rule are taken and the Recognize-Act cycle is again repeated.

OPS5 source code (the "rules") is compiled into VAX assembly code ("Threaded code") which is assembled and linked with the OPS5 kernel to create a stand-alone executable image. Since each OPS5 application is linked into an executable image, OPS5 allows each application to also be linked to external subroutines written in any supported VAX language. This provides an OPS5 application with complete access to the VMS system services and the ability to interact with external tasks.

F.1.2.1 OPS5 Evaluation Tests -

Prior to deciding to using OPS5 for developing EDMs, its ability to implement basic functions we would have to perform within an EDM was evaluated. To do this we wrote OPS5 applications to evaluate its ability to

1. Interface with external subroutines. The external subroutines would be used to read the Named Knowledge elements, receive command lines from the shell and pass commands back to the shell for execution.

2. Suggest fringe processing steps to take based on a collection of heuristic fringe processing rules used by operators who process fringes.

The test applications were developed with little difficulty and performed efficiently confirming the belief that OPS5 is the proper language for use in developing the FAS EDMs.

F.2 Design Considerations For Developing A FAS EDM

The architecture for a Fringe Analysis EDM has not yet been designed. However, a number of features seem reasonable to incorporate in the final design of an EDM.

F.2.1 EDM Scope

The scope of each EDM is to be limited to a narrow scope of expertise. By limiting each EDM to a small rule base, development of each EDM will be faster and easier to maintain. If the knowledge to make a decision falls outside of the knowledge boundaries of a given EDM, a valid action to take is for the primary EDM to invoke an second EDM possessing knowledge in a different area. In this event, the primary EDM may either choose to exit and pass control to the secondary EDM (which then becomes the primary) or to wait for the secondary EDM to exit and use the knowledge gained by the secondary EDM for making subsequent decisions.

F.2.2 Implementation Languages

The EDMs will be written in OPS5 with operating system interface subroutines written in either in Fortran or VAX Basic as appropriate.

F.2.3 EDM/Shell Interface

The possible EDM/Shell interactions are identical to the interactions any other module can have in the FAS. However, once an EDM is activated, it controls the flow of processing in the system by repeatedly passing back commands to the Shell to be dispatched to a lower level subprocess. In effect the EDM becomes a "virtual" FAS operator.

End of Document